A modeling language for novice engineers to design well at SaaS product companies

Mrityunjay Kumar
IIIT Hyderabad
Hyderabad, Telangana, India
mrityunjay.k@research.iiit.ac.in

Venkatesh Choppella
IIIT Hyderabad
Hyderabad, Telangana, India
venkatesh.choppella@iiit.ac.in

ABSTRACT

Software-as-a-Service (SaaS) product companies have brought in significant changes in how we build software from architecture and engineering process perspective. SaaS products are large, distributed software systems hosted in cloud and built using collaborating services (or micro-services). The software releases happen in days and weeks, necessitating an agile development process. Novice engineers (those who join the company fresh from college) need to become comfortable with complex systems and proficient in agile delivery with high quality, otherwise they fall behind in productivity.

The paper posits that, to be successful at these SaaS product companies, the novice engineers need good modeling and design skills. While this has been for all software development, the changes driven by SaaS products have made this need more acute. Such skills will allow them to capture their feature behaviors (in context of their understanding of the larger product) in an implementation-independent manner and any knowledge gaps can be identified and bridged by their collaborators.

We propose a modeling language that is easy for them to learn and use, and which has characteristics suitable for the kind of engineering work they need to do in their early years in a SaaS product company. This modeling language is based on the notion of Transition Systems. The paper demonstrates the usage and value of this language by creating a model for a real feature. The modeling language is quite general and transcends abstraction boundaries. We also present a modeling process that should be used with this language for better results.

This is a short position paper that presents an idea about a new modeling language for a specific purpose (helping novice engineers design well at SaaS product companies). Validation studies for the language and the design process is a work in progress and the results will be shared in a full paper later.

CCS CONCEPTS

• **Software and its engineering** → **System modeling languages**; *Software design engineering*; • **Social and professional topics** → *Software engineering education.*

ISEC 2023, February 23–25, 2023, Allahabad, India

ACM ISBN 979-8-4007-0064-4/23/02…$15.00
https://doi.org/10.1145/3578527.3578548

KEYWORDS

Software Modeling Language, SaaS product design, Novice Engineer, Transition Systems

ACM Reference Format:
Mrityunjay Kumar and Venkatesh Choppella. 2023. A modeling language for novice engineers to design well at SaaS product companies. In *16th Innovations in Software Engineering Conference (ISEC 2023), February 23–25, 2023, Allahabad, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578527.3578548

1 INTRODUCTION

SaaS [5] is changing the development ecosystem. Modern software product companies are very different from companies a few decades ago, both in terms of system architecture and engineering processes. Many software systems are today delivered in Software-as-a-Service (SaaS) model. A SaaS product is a large, interactive system that delivers its functionality through a set of services (or micro-services). A typical SaaS product is a distributed, multi-tenant system using multiple external APIs and a service-oriented architecture. SaaS is now an established way of building and delivering software products.

SaaS product users expect quick turnaround (given the software is on cloud) when they discover defects or desire new capabilities. Also, given multi-tenancy (multiple customers are served using same software system and code), small defects can have significant impact across the customer base. Even a simple feature needs to be engineered with rigor. These needs have driven SaaS product companies to rely on Agile methodologies to shrink their release cycles to days and weeks and employ processes like Continuous Integration and Delivery (CI/CD) and roles like DevOps. This also drives the need for the engineers to be all-in-one and know-it-all: performing development, testing, deployment, and production support tasks for the features they build.

Novice engineers[1] *have skill and knowledge gaps in two key areas.* Given the architectural and process needs of SaaS products described above, novice engineers have two key skill knowledge gaps when they join the company: lack of cloud-first development skills, and design skills. Skills required for cloud-first development have not yet found their way into regular computer science curricula [13]. We also find that novice engineers do not demonstrate good skills in designing and capturing their design in written form [22]. Gaps in cloud and design skills mean that the onboarding time required can be quite high.

Onboarding cycles are short and not enough. The on-boarding journey of novice engineer across product companies are similar

[1]We use the term novice engineer to refer to fresh campus hires as well as final year students ready to join a company.

[11, 18, 19] and consist of three parts: 1) provide some training and orientation about tools, technologies and processes used by the engineering teams, 2) get subject matter experts (SMEs) deliver a few sessions about the product and business, and 3) point them to code repository and some partial documentation to start exploring the product. These are usually done in a few weeks, and then simple engineering tasks like debugging and simple defect fixes are assigned to familiarize them with the product and process, followed by new feature development. Such a short onboarding cycle (only several weeks) means novice engineers carry their knowledge gap to their feature work.

Engineers will benefit from a design-first approach. Incomplete knowledge of the product and architecture (due to short onboarding cycle) can cause production issues. Our proposed way to alleviate such a situation is for the engineers to put together a design of the feature (including a model of the feature behavior and dynamics vis-a-vis the larger product) and share with their mentors or seniors so that their knowledge gaps can be identified, and designs updated. Development based on such a reviewed model and design is expected to lead to a higher-quality implementation.

In this paper, we propose a modeling language derived from dynamical systems theory, which is the notion of transition systems that can be used for this purpose. We also present a modeling process to make the usage of the language more effective.

A note on our usage of the word 'model'. We use model as a higher level of abstraction than an implementation design. Using Fowler's terminology [14], we use something between 'model as a sketch' and 'model as a blueprint'.

Section 2 presents related work. Section 3 introduces Transition Systems as the modeling language we propose. Section 4 applies the modeling language and the process to a simple but real feature work to illustrate the ease and value of the language. Section 5 concludes and discusses future work planned.

2 RELATED WORK

Development model is changing. SaaS products bring a different development model than traditional software [30]. They are designed for cloud deployment [32], use multi-tenancy [21], and usually require a different approach to architecture and design [3]. SaaS has also impacted the development process. Agile [8], sometimes with variation [1] is the preferred development process due to faster release promise, and Continuous Integration and Delivery (CI/CD) is expected from all engineering teams [27]. These changes mean that the expectations from the engineers, esp. novice engineers are changing as well, and they are expected to understand the system and end to end while building features. Having a good mental model of the entire system and applying it when building a feature is going to be key to novice engineers' effectiveness. Sterman [29] says "[In brain,] active modeling occurs well before sensory information reaches the areas of the brain responsible for conscious thought", and Morecroft [26] suggests that a mental model can be augmented by a formal model to expedite the learning. In this sense, modeling and designing a solution is nothing but a tool to enhance the mental model and extend the cognition.

Curriculum has not kept pace. Shaw et al. [28] talked about 'changing face of software' in 2005 and listed down the needs of modern software development and how software engineering education needs to address it. Curriculum hasn't necessarily kept pace; for example, AICTE, the regulatory body for technical education in India doesn't include Software Engineering courses as core courses for Computer Science and Engineering model curriculum [2] and many colleges in India do not include this in their coursework. This creates a challenge for the novice engineers.

UML may not be right for novice engineers. UML is the most prevalent modeling language in teaching and industry use, a de-facto standard [12]. However, [6] presented data from survey in two phases of software practitioners and reported decrease in participants' satisfaction with modeling tools. They also reported that the use of UML as the dominant tool went down (from 51% very often to 33.4% very often). [7] show that "students' perception of the value of modeling declines as they progress in their education" and wonder if this may be because UML is not suitable for the kind of problems they face. Li et al. [23] analyzed 2 large open-source software systems and showed that more than 80% of defects are semantic (functionality related) in nature, hence the modeling language that captures the dynamics should help. However, Moisan and Rigault [25] reported (about their students from college and professionals) that while Class and Object diagrams of UML are well accepted, State models and Sequence diagrams are not. Ten years later, Koç et al. [20], in their 2019 systematic literature review of research trends on UML diagrams pointed out that "...sequence and state machine diagrams had the low rate of usage". UML doesn't seem to be the tool of choice for modeling the dynamics. In addition, the characteristics we need in such a language (easy to learn and use, create models quickly and refine often, be able to reason with them and analyze or simulate the behavior) go beyond what UML offers.

3 TRANSITION SYSTEMS AS A MODELING LANGUAGE

We use the notion of transition systems [10, 31] and present its usage as a lightweight modeling language for novice engineers to use. It should be noted that the intent here is to adapt a formal system (from a well-defined and rich domain of dynamical systems and formal modeling) and make it useful in the context of software engineering practices by novice engineers (who typically do not get exposed to formal modeling). We will forego some of the rigor to achieve easy of learning and use.

3.1 Transition System definition

A transition system is a six tuple $\{X, X^0, U, f, Y, h\}$ where X is the set of states, X^0 is the set of initial states, U is the set of actions, f is the transition relation which is a subset of $(X \times U) \times X$, Y is the set of observables (or output space), and h is a *display map*, mapping states to observables, $y = h(x)$.

The transition system models the dynamics of the system as depicted in Figure 1.

When there are more than a few states or actions, the transition function f is better represented as a table. T is the transition table corresponding to transition function f and t_{ij} represents the dynamics when action u_j is triggered in state x_i for the system - $f(x_i, u_j)$. We will use t_{ij} and f_{ij} interchangeably, $t_{ij} = f_{ij} = f(x_i, u_j)$.

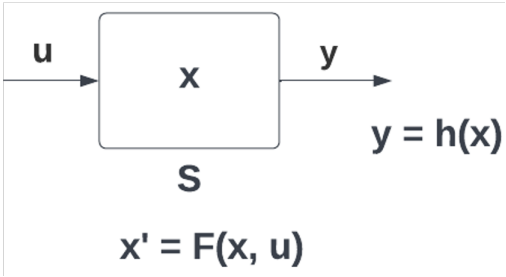

Figure 1: Transition System S (x: state, u: action, y: observable, h: display map, F: transition relation)

3.2 An Example

Let's say that we want to add a Like Icon for the product details page in an e-commerce site. When the icon is clicked, it goes from Liked to NotLiked, or vice versa.

Transition System Model: For the six tuple $\{X, X^0, U, f, Y, h\}$ for this system,

$X = \{Liked, NotLiked\}$,
$X^0 = \{NotLiked\}$,
$U = \{click\}$,
$Y = \{Redheart, Whiteheart\}$

Redheart and Whiteheart are the icons used to show Liked and NotLiked state respectively.

The transition function can be shown in a table, but given it is a simple system with 2 states and one event, we can describe it right here: $f(Liked, Click) = NotLiked$, $f(NotLiked, Click) = Liked$. The display map h can be described easily too: $h(Liked) = Redheart$, $h(NotLiked) = Whiteheart$.

3.3 Why a new modeling language, why not UML?

As described in Section 1, we want novice engineers in SaaS product companies to use a modeling language which they can use to create models of their system's behavior quickly. UML does not fit well in our context (Section 2), transition systems as a modeling language fit better: 1) it eases creation and refinement of models - the vocabulary is small and the notation is simple, 2) it brings system-focus and a high level of abstraction, allowing better mapping to specification, and 3) its core concepts come from Theory of Computation [9] and Discrete Mathematics [17] courses which engineers are likely to be already familiar with. This allows them to construct the meaning for themselves [4], since they can connect this new knowledge of modeling language to their prior knowledge [15, 16, 24].

3.4 Modeling Process

It is not enough to have a modeling language. It is also important to have a modeling process that enables capturing the richness of the system behavior in the model. We present a modeling process to go with our modeling language.

(1) Specify action space U: Inputs in the specification map to list of actions.
(2) Specify the output space Y: Outputs in the specification map to list of observables
(3) Using Y and U and the specification provided for the system, identify the states the system should maintain. This will be an initial estimate and will get updated in further iterations. Let's call this set of states A.
(4) Identify the minimum set of information that need to be held by each state in A to enable the set of observables Y. Use this, along with A, to define X.
(5) Define the display map h based on X and Y.
(6) Define the set of initial states X^0 based on the problem specification.
(7) Using U, X, and the specification, define the transition relation (f) in the form of transition table T. This will also update the set of information required to be held in the state, so update X as appropriate.
(8) Specify U and Y in terms of UX (or other interface definitions) provided as part of the specification.
(9) Write the system test cases to cover the behavior captured in the transition table T and user test cases using action space U and display map h.

The engineer needs to discuss and review this model with the stakeholders and ensure system behavior is as expected. This will lead to refining X, X^0, U, f, Y, and h through iterative application of steps above which is desirable and expected.

3.5 Applicability and Limitations

As mentioned before, the goal of this language is to capture the dynamics (behavior) of the system succinctly and with precision so that novice engineers can use it to become more effective. While the language is generic, our scope in this paper is to address the kind of feature development a novice engineer deals with. Here are a few limitations to keep in mind: a) Application software systems, which usually implement business and consumer logic and are usually interactive, are good candidates for using this language, systems heavy on computations or specific algorithm not so much. b) We have not considered the design of complex features where state space can be very large and make the model unmanageable. Similarly, we have not considered modeling system of systems. We do not expect novice engineers to build such systems right away. c) The vocabulary used in our language is small. However, any modeling exercise also requires domain vocabulary, and that is likely to be large, and so the novice engineer still must deal with a large vocabulary when modeling.

4 MODELING A SIMPLE APPLICATION

To illustrate the power and simplicity of creating models using transition systems, this section walks through the lifecycle of a typical feature work assigned to a novice engineer in a SaaS product company and illustrates model creation and modification cycles.

In this illustration[2], E1 is a retailer whose flagship product is their ecommerce site from where they sell products. The novice engineer is assigned a simple feature to build and will own its lifecycle going forward, which includes handling enhancements and defects reported by customers.

[2]We have skipped or merged a few steps from the modeling process (as presented in Subsection 3.4) to keep this succinct.

The product manager (PM) will have written a specification for the feature, along with a User Experience Design (UX) specification, something like this (more detailed in real project):

The goal of this feature is to allow users to see product videos and get more interested in buying the product. A new tab called 'Videos' should be available on the product details page. When user navigates to this tab, it should show a set of user videos available for this product. User can select a video to be played in the video playing area at the bottom of this tab. User should be able to stop a playing video and select any other video they like to play.

Model creation: We need to define the transition system six tuple $\{X, X^0, U, f, Y, h\}$.

Actions *(U)*: The behavior is clear from the feature description. We start with listing down the actions that can be performed on this system (U). Three key actions are: 1) Load the tab, 2) Select the video to play, and 3) Stop the playing video. So, we can define $U = \{Load, Select, Stop\}$.

Observables *(Y)*: The expected output (y) is a function (h) of state(x). Based on the specification, the observables are: 1) A page with video list and no video selected to play, and 2) the page with selected video playing (or stopped). Hence the information that is required to be part of the state is: 1) Video List, 2) Selected Video identifier, 3) Video Player reference.

States *(X, X^0)*: U and Y suggest that the system should support three states: 1) $Initial$ (The tab is not yet loaded), 2) $LoadedTab$ (The tab is loaded and is ready to play) and 3) $PlayingVideo$ (A video has been selected and is playing). When a playing video is stopped, the system goes to $LoadedTab$ state. The system states need to keep track of the following information at the minimum (some of this come from needs of Y, while others come from processing needs). We also include (in square bracket) what kind of data type this will be.

(1) Product identifier ($Integer\ i$)- Current product reference
(2) State Name ($Enum\ n$)- {Initial, LoadedTab, PlayingVideo}
(3) Video List (l) ($VideoObject[]\ l$)- List of videos and their details
(4) Selected video ($Integer\ s$)- Reference to the currently selected video
(5) Video Player reference ($PlayerRef\ p$) - Handle to the video player to render and play video

Thus, each state x is a five tuple (i, n, l, s, p). It starts with the initial state $x^0 = (-1, Initial, null, -1, null)$.

Display Map (h): The expected output (y) is a function (h) of state(x). As identified above, the observable needs to be able to show the videos, distinguish the selected video in some way, and show the player. Hence h should extract l, s, and p from the state.

So y is a three tuple (a, b, c) where $(a, b, c) = h(x), x = (i, n, l, s, p)$ and $a = l, b = s, c = p$

These values will be used appropriately to render the UX provided by the PM. The display map is defined in Table 2b.

Transition relation *(f)*: Using U, X and behavior description, we can define the transition relation f. The transition table that represents the relation f is shown in Table 2a. In the table, F1, F2, and F3 are shorthand to refer to the logic applied by the transition function when the specific u and x are present: $f(Initial, Load) = F1$, $f(LoadedTab, Select) = F2$, $f(PlayingVideo, Select) = F2$, and $f(PlayingVideo, Stop) = F3$.

There are 3 key procedures in the table which are separately described in Table 1.

Table 1: Function Behavior (State and Action names are in italics)

Behavior
F1 When *Load* Action is applied to the State $Initial$, the system needs to initialize, which means state is set to the initial state. ***State Update***: $(i', n', l', s', p') = f(x), x = (i, n, l, s, p)$ where $i' = getProductId()$, $n' = LoadedTab$, $l' = getVideoList(i')$, $s' = -1$, $p = getVideoPlayerRef(i')$. The product id is populated (i), state name is set *Loaded* (n), the list of videos for the product is fetched and added to state (l), selected video is set to -1 to denote no selection (s), a video player is initialized, and its reference is added to state (p).
F2 When *Select* Action is applied to the State $LoadedTab$ or $PlayingVideo$, the system needs to start playing the selected video and state variables get updated. ***State Update***: $(i', n', l', s', p') = f(x), x = (i, n, l, s, p)$ where $i' = i$, $n' = PlayingVideo$, $l' = l$, $s' = currentSelection()$, $p' = p$. ***Other Actions***: $playSelectedVideo(p', s')$ There is no change to product id, list of videos or player reference. Selected Video is updated to the new value (s') and state name is updated to $PlayingVideo$ (n'). Additionally, the newly selected video is played.
F3 When *Stop* Action is applied to the State $PlayingVideo$, the system needs to stop playing the selected video and state variables get updated. ***State Update***: $(i', n', l', s', p') = f(x), x = (i, n, l, s, p)$ where $i' = i$, $n' = LoadedTab$, $l' = l$, $s' = s$, $p' = p$. ***Other Actions***: $stopVideo(p)$ There is no change to any state variable except state name which is updated to $LoadedTab$ (n'). Additionally, the currently selected video is stopped.

Table 2: Transition table and Display map for Video feature

	ACTION		
STATE	*Load*	*Select*	*Stop*
Initial	F1	X	X
LoadedTab	X	F2	X
PlayingVideo	X	F2	F3

(a) Transition table (X denotes an undefined transition)

STATE	*Observable*
Initial	X
LoadedTab	Rendered Page
PlayingVideo	Selected Video Playing

(b) Display map (X denotes undefined observable)

4.1 Benefits of this model

As can be seen, the dynamics of the system is clearly captured in the procedures F1, F2, and F3, in an implementation-independent language. This makes it easy for the Product Manager (PM) to review the planned implementation for requirement gaps. For example, the behavior when a new video is clicked while old one is still playing is ambiguous in spec and can be identified and discussed during review of F2.

The input and output (which determine the user experience) are captured in U, Y and h. This separation makes it easy to review various user-facing aspects of the feature (which is a big area where

defects arise in production [20]) and do quick prototyping of the interfaces if necessary.

When the feature is released and customer starts using the feature, the model can be used to analyze the feedback from the production usage and localize them to specific parts of the model (the details of analysis are not included due to space constraints, they can be seen in the examples of product change requests).

5 CONCLUSION AND FUTURE WORK

New Engineers at a SaaS product companies find themselves in somewhat unique situation - they need to build high-quality features within a large system in a short time using partial knowledge of product they gained during short onboarding period. We make a case for them to use modeling and design to achieve their goals, using a modeling language and process that is systems-focused, easy to learn, and enables rapid creation and iteration. A Transition System can model a system simply through a six-tuple $\{X, X^0, U, f, Y, h\}$, and is proposed as a modeling language.

By illustrating the modeling through a real application, we have shown how the usage of this language can help achieve the key design characteristics and engineering needs of a feature: a) **Clear interface separation** - Interface is clearly separated from core logic by using observables and actions. b) **Modular behavior design** - Transition table allows us to modularize the logic for each state-action combination. Common behavior can be reused. c) **Extensibility through states** - Since the behavior is completely determined by the current state and current action, new behaviors can be added by adding new states or actions. d) **Implementation modularity** - Transition table and display map partition the design into distinct implementation chunks which makes the implementation quicker and modular. e) **Excellent test coverage** - Since entire behavior is codified in the transition table and display map, creating test cases is straightforward and comprehensive.

Future work: This new idea needs to be validated in the field. We plan to run experiments with students from multiple colleges by teaching them the language and the process and measure the impact on their designing and development skills. We also plan to work with new campus hires from a few SaaS companies to check the efficacy. To illustrate the benefits of the modeling language better, we will work out the modeling of toy features in both Transition Systems language as well as in UML.

REFERENCES

[1] Puneet Agarwal. 2011. Continuous SCRUM: agile management of SAAS products. In *Proceedings of the 4th India Software Engineering Conference*. 51–60.

[2] AICTE. 2018. MODEL CURRICULUM UNDERGRADUATE DEGREE IN ENGINEERING & TECHNOLOGY Vol 1. (2018), 340–371.

[3] Saiqa Aleem, Rabia Batool, Faheem Ahmed, and Asad Masood Khattak. 2018. Design guidelines for SaaS development process. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 825–831.

[4] Roya Jafari Amineh and Hanieh Davatgari Asl. 2015. Review of constructivism and social constructivism. *Journal of Social Sciences, Literature and Languages* 1, 1 (2015), 9–16.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (apr 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[6] Omar Badreddin, Rahad Khandoker, Andrew Forward, Omar Masmali, and Timothy C Lethbridge. 2018. A decade of software design and modeling: A survey to uncover trends of the practice. In *Proceedings of the 21th acm/ieee international conference on model driven engineering languages and systems*. 245–255.

[7] Omar Badreldin, Timothy Lethbridge, Arnon Sturm, Waylon Dixon, Abdelwahab Hamou-Lhadj, and Ryan Simmons. 2015. The effects of education on students' perception of modeling in software engineering. CEUR Workshop Proceedings.

[8] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. The agile manifesto.

[9] Somenath Biswas. 2022. NPTEL Syllabus - Theory of Computation. https://archive.nptel.ac.in/content/syllabus_pdf/106104028.pdf [Online; accessed 15-September-2022].

[10] Venkatesh Choppella, Kasturi Viswanath, and Mrityunjay Kumar. 2021. Algodynamics: Algorithms as systems. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[11] Barthélémy Dagenais, Harold Ossher, Rachel KE Bellamy, Martin P Robillard, and Jacqueline P De Vries. 2010. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 275–284.

[12] Andrew Forward, Omar Badreddin, and Timothy C Lethbridge. 2010. Perceptions of software modeling: a survey of software practitioners. In *5th workshop from code centric to model centric: evaluating the effectiveness of MDD (C2M: EEMDD)*.

[13] Derek Foster, Laurie White, Joshua Adams, D Cenk Erdil, Harvey Hyman, Stan Kurkovsky, Majd Sakr, and Lee Stott. 2018. Cloud computing: developing contemporary computer science curriculum for a cloud-first future. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 130–147.

[14] Martin Fowler. 2017. UML mode, 2003. url: http://www. martinfowler. com/bliki. *UmlMode. html* (2017).

[15] Robert Mills Gagne et al. 1965. The conditions of learning.

[16] Robert M Gagne, Walter W Wager, Katharine C Golas, John M Keller, and James D Russell. 2005. Principles of instructional design. *Performance Improvement* 44, 2 (2005), 44–46.

[17] SUDARSHAN IYENGAR and Prabuchandran K.J. 2022. NPTEL Syllabus - Discrete Mathematics. https://archive.nptel.ac.in/content/syllabus_pdf/106106183.pdf [Online; accessed 15-September-2022].

[18] Maggie Johnson and Max Senges. 2010. Learning to be a programmer in a complex organization: A case study on practice-based learning during the onboarding process at Google. *Journal of Workplace Learning* (2010).

[19] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. 2021. A case study of onboarding in software teams: Tasks and strategies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 613–623.

[20] Hatice Koç, Ali Mert Erdoğan, Yousef Barjakly, and Serhat Peker. 2021. UML diagrams in software engineering research: a systematic literature review. *Multidisciplinary Digital Publishing Institute Proceedings* 74, 1 (2021), 13.

[21] Rouven Krebs, Christof Momm, and Samuel Kounev. 2012. Architectural concerns in multi-tenant saas applications. *Closer* 12 (2012), 426–431.

[22] Mrityunjay Kumar and Venkatesh Choppella. 2022. A Study of the Design and Documentation Skills of Industry-Ready CS Students*. In *COMPUTE 2022* (Jaipur, India) *(COMPUTE 2022)*. Association for Computing Machinery, New York, NY, USA, 23–28. https://doi.org/10.1145/3561833.3561842

[23] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 25–33.

[24] M David Merrill. 2002. First principles of instruction. *Educational technology research and development* 50, 3 (2002), 43–59.

[25] Sabine Moisan and Jean-Paul Rigault. 2009. Teaching Object-Oriented Modeling and UML to Various Audiences. In *Proceedings of the 2009 International Conference on Models in Software Engineering* (Denver, CO) *(MODELS'09)*. Springer-Verlag, Berlin, Heidelberg, 40–54. https://doi.org/10.1007/978-3-642-12261-3_5

[26] John Morecroft. 2004. Mental models and learning in system dynamics practice. *Systems modelling: Theory and practice* (2004), 101–126.

[27] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 21–30.

[28] Mary Shaw, Jim Herbsleb, and Ipek Ozkaya. 2005. Deciding what to design: Closing a gap in software engineering education. In *Proceedings of the 27th international Conference on Software Engineering*. 607–608.

[29] John D Sterman. 1994. Learning in and about complex systems. *System dynamics review* 10, 2-3 (1994), 291–330.

[30] WeiTek Tsai, XiaoYing Bai, and Yu Huang. 2014. Software-as-a-service (SaaS): perspectives and challenges. *Science China Information Sciences* 57, 5 (2014), 1–15.

[31] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of modeling and simulation*. Academic press.

[32] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. 2010. Cloud computing research and development trend. In *2010 Second international conference on future networks*. IEEE, 93–97.